

Perspectives in Certificate Translation

Gilles Barthe and César Kunz

IMDEA Software, Spain

Abstract. Certificate translation is a general mechanism to transfer evidence across abstraction layers, from source code to executable code. We review the general principles behind certificate translation and the main results achieved so far, and outline directions for future work.

1 Introduction

Certificate translation aims to provide the benefits of (typically interactive) source code verification to code consumers, building upon the notion of certificate used in Proof Carrying Code [14]. More precisely, the primary goal of certificate translation is to transform certificates of source programs into certificates of compiled programs. By design, certificate translation is very general and can be used to enforce arbitrarily complex properties of programs, provided they can be expressed and formally established using source code verification frameworks.

The problem of certificate translation can be expressed informally in a very general form. Consider two programming languages, a source language Prog_s and a target language Prog_t , each equipped with a specification language, respectively Spec_l and Spec_t , and with a verification framework. We assume that the verification frameworks are equipped with a notion of proof object, a.k.a. certificate, and axiomatize the verification frameworks as ternary relations for certificate checkers, written $c : p \models s$, stating that c is a certificate that p adheres to s , where p is a program belonging to Prog_s (resp. Prog_t), s is a specification belonging to Spec_s (resp. Spec_t) and c belongs to the set Prf_s of source certificates (resp. Prf_t of target certificates). Assuming a compiler for programs $\mathcal{C} : \text{Prog}_s \rightarrow \text{Prog}_t$, and a compiler for specifications $\mathcal{C}_{\text{spec}} : \text{Spec}_s \rightarrow \text{Spec}_t$, the problem is to find a certificate translation, i.e. a function $\mathcal{C}_{\text{cert}} : \text{Prf}_s \rightarrow \text{Prf}_t$ such that for all source programs p , policies s , and certificates c ,

$$c : p \models s \implies \mathcal{C}_{\text{cert}}(c) : \mathcal{C}(p) \models \mathcal{C}_{\text{spec}}(s)$$

The formal study of certificate translation requires that all the parameters are instantiated, and defined formally: source and target programming and specification languages, certificate checkers for source and compiled programs, and finally compilers for programs, specifications, and certificates. Our work focuses on verification infrastructures that rely on verification condition generators (VC Generator), which are used in many interactive verification environments at source level, and in automated verification tools at compile level. A VC Generator can be seen

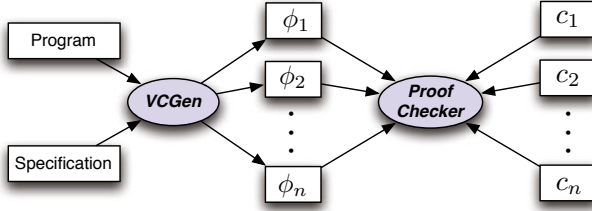


Fig. 1. Verification Infrastructure

as a strategy of applying the rules of an Hoare logic, and extracts automatically a set of proof obligations from an annotated program—annotations include loop invariants, preconditions and postconditions. There are several advantages to verification condition generators, other than their predominance in verification tools: the proof obligations are expressed in the specification language, abstracting away from the programming language; moreover, the certificate does not need to store the application of the rules of the Hoare logic, since the strategy is fixed; hence, standard notions of proof objects can be used as certificates. Figure 1 considers a program verification infrastructure for a specific programming language. The VC Generator takes a program and specification and produces a set of proof obligations attesting the adherence of the program to its specification. A certificate c consists of a set of proof objects $(c_i)_{i \in I}$ such that each proof obligation generated by the VC Generator is validated by a proof object c_i .

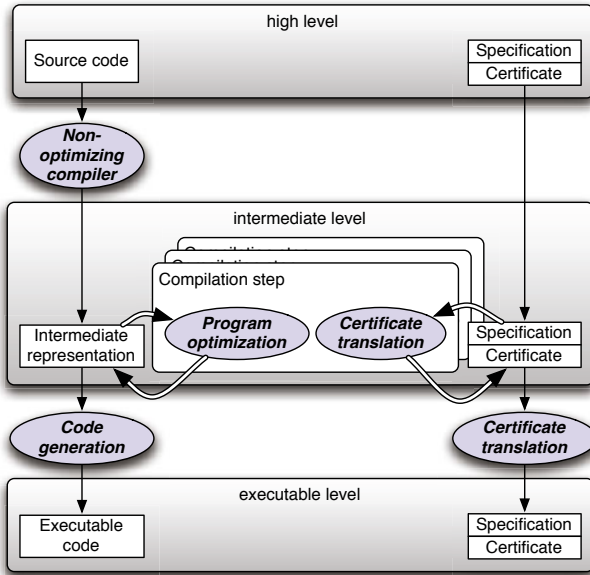


Fig. 2. Overall picture of the Optimizing Infrastructure